Lecture 2
Introduction to Semantics
Semantics

**Question**: What is the meaning of a program?
Semantics

**Question:** What is the meaning of a program?

**Answer:** We could execute the program using an interpreter or a compiler, or we could consult a manual...

...but none of these is a satisfactory solution.
Formal Semantics

Three Approaches

• Operational
  ▶ Model program by execution on abstract machine
  ▶ Useful for implementing compilers and interpreters

  $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

• Denotational:
  ▶ Model program as mathematical objects
  ▶ Useful for theoretical foundations

  $[e]$

• Axiomatic
  ▶ Model program by the logical formulas it obeys
  ▶ Useful for proving program correctness

  $\vdash \{\phi\} e \{\psi\}$
Arithmetic Expressions
Syntax

A language of integer arithmetic expressions with assignment.
Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

\[ x, y, z \in \text{Var} \]
\[ n, m \in \text{Int} \]
\[ e \in \text{Exp} \]
Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

\[ x, y, z \in \text{Var} \]
\[ n, m \in \text{Int} \]
\[ e \in \text{Exp} \]

BNF Grammar:

\[ e ::= x \]
\[ \mid n \]
\[ \mid e_1 + e_2 \]
\[ \mid e_1 \times e_2 \]
\[ \mid x := e_1 ; e_2 \]
Ambiguity

What expression does the string “1 + 2 * 3” describe?
Ambiguity

What expression does the string “1 + 2 * 3” describe?

There are two possible parse trees:
Ambiguity

What expression does the string “1 + 2 * 3” describe?
There are two possible parse trees:

```
+  
1  
*  
2  
3
```

In this course, we will distinguish **abstract syntax** from **concrete syntax**, and focus primarily on abstract syntax (using conventions or parentheses at the concrete level to disambiguate as needed).
Representing Expressions

BNF Grammar:

\[ e ::= x \]
\[ n \]
\[ e_1 + e_2 \]
\[ e_1 * e_2 \]
\[ x := e_1 ; e_2 \]
Representing Expressions

BNF Grammar:

\[ e ::= x \]
\[ n \]
\[ e_1 + e_2 \]
\[ e_1 \times e_2 \]
\[ x := e_1 ; e_2 \]

OCaml:

```ocaml
type exp = Var of string
| Int of int
| Add of exp * exp
| Mul of exp * exp
| Assgn of string * exp * exp
```

Example: Mul(Int 2, Add(Var "foo", Int 1))
Representing Expressions

BNF Grammar:

\[
e ::= x
def n
def e_1 + e_2
def e_1 * e_2
def x := e_1 ; e_2
\]

Java:

```java
abstract class Expr {}
class Var extends Expr { String name; ... }
class Int extends Expr { int val; ... }
class Add extends Expr { Expr exp1, exp2; ... }
class Mul extends Expr { Expr exp1, exp2; ... }
class Assgn extends Expr { String var, Expr exp1, exp2; ... }
```

Example: new Mul(new Int(2), new Add(new Var("foo"), new Int(1)))
Quiz

- $7 + (4 \times 2)$ evaluates to ...?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
Quiz

• $7 + (4 \times 2)$ evaluates to 15
• $i := 6 + 1 ; 2 \times 3 \times i$ evaluates to ...?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1; 2 \times 3 \times i$ evaluates to 42
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1; 2 \times 3 \times i$ evaluates to 42
- $x + 1$ evaluates to ...?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1 ; 2 \times 3 \times i$ evaluates to 42
- $x + 1$ evaluates to error?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1 ; 2 \times 3 \times i$ evaluates to 42
- $x + 1$ evaluates to error?

The rest of this lecture will make these intuitions precise...
Mathematical Preliminaries
Binary Relations

The \textit{product} of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$. 
Binary Relations

The \textit{product} of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$.

A \textit{binary relation} on $A$ and $B$ is just a subset $R \subseteq A \times B$. 
Binary Relations

The *product* of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$.

A *binary relation* on $A$ and $B$ is just a subset $R \subseteq A \times B$.

Given a binary relation $R \subseteq A \times B$, the set $A$ is called the *domain* of $R$ and $B$ is called the *range* (or *codomain*) of $R$. 
Binary Relations

The *product* of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$.

A *binary relation* on $A$ and $B$ is just a subset $R \subseteq A \times B$.

Given a binary relation $R \subseteq A \times B$, the set $A$ is called the *domain* of $R$ and $B$ is called the *range* (or *codomain*) of $R$.

**Some Important Relations**

- empty: $\emptyset$
- total: $A \times B$
- identity on $A$: $\{(a, a) \mid a \in A\}$.
- composition $R; S$: $\{(a, c) \mid \exists b. (a, b) \in R \land (b, c) \in S\}$
A *(total) function* $f$ is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$. 

![Diagram of functions and non-functions](image)
Functions

A *(total)* function $f$ is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When $f$ is a function, we usually write $f : A \to B$ instead of $f \subseteq A \times B$. 
Functions

A (total) function $f$ is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When $f$ is a function, we usually write $f : A \rightarrow B$ instead of $f \subseteq A \times B$.

The image of $f$ is the set of elements $b \in B$ that are mapped to by at least one $a \in A$. Formally:

$$\text{image}(f) \triangleq \{ f(a) \mid a \in A \}$$

$$f(x) = x \times 0$$
Some Important Functions

Given two functions $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition of $f$ and $g$ is defined by: $(g \circ f)(x) \triangleq g(f(x))$  

Note order!
Some Important Functions

Given two functions \( f : A \rightarrow B \) and \( g : B \rightarrow C \), the composition of \( f \) and \( g \) is defined by: \( (g \circ f)(x) = g(f(x)) \)  

Note order!

A partial function \( f : A \rightarrow B \) is a total function \( f : A' \rightarrow B \) on a set \( A' \subseteq A \). The notation \( \text{dom}(f) \) refers to \( A' \).

\[
f \subseteq A' \times B \subseteq A \times B
\]
Some Important Functions

Given two functions $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition of $f$ and $g$ is defined by: $(g \circ f)(x) = g(f(x))$ \hspace{1cm} \text{Note order!}

A partial function $f : A \rightarrow B$ is a total function $f : A' \rightarrow B$ on a set $A' \subseteq A$. The notation $\text{dom}(f)$ refers to $A'$.

A function $f : A \rightarrow B$ is said to be injective (or one-to-one) if and only if $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$. 
Some Important Functions

Given two functions $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition of $f$ and $g$ is defined by: $(g \circ f)(x) = g(f(x))$ \hspace{1cm} \text{Note order!}

A partial function $f : A \rightarrow B$ is a total function $f : A' \rightarrow B$ on a set $A' \subseteq A$. The notation $\text{dom}(f)$ refers to $A'$.

A function $f : A \rightarrow B$ is said to be \textit{injective} (or \textit{one-to-one}) if and only if $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$.

A function $f : A \rightarrow B$ is said to be \textit{surjective} (or \textit{onto}) if and only if the image of $f$ is $B$. 
Operational Semantics
Overview

An operational semantics describes how a program executes on some abstract (imaginary) machine.
Overview

An operational semantics describes how a program executes on some abstract (imaginary) machine.

A small-step semantics describes how such an execution proceeds from configuration to configuration: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)
Overview

An operational semantics describes how a program executes on some abstract (imaginary) machine.

A small-step semantics describes how such an execution proceeds from configuration to configuration: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)

For our language, a configuration \( \langle \sigma, e \rangle \) is a pair of:

- a store \( \sigma \) that records the values of variables,
- and the expression \( e \) being evaluated.
Overview

An operational semantics describes how a program executes on some abstract (imaginary) machine.

A small-step semantics describes how such an execution proceeds from configuration to configuration: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)

For our language, a configuration \( \langle \sigma, e \rangle \) is a pair of:

- a store \( \sigma \) that records the values of variables,
- and the expression \( e \) being evaluated.

More formally:

\[
\sigma \in \text{Store} \triangleq \text{Var} \rightarrow \text{Int} \\
\text{Config} \triangleq \text{Store} \times \text{Exp}
\]

(A store is a partial function from variables to integers.)
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of \( \text{Config} \times \text{Config} \).
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

**Notation:** $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

which means $(\langle \sigma, e \rangle, \langle \sigma', e' \rangle) \in \rightarrow$.

\[
\rightarrow((\langle \sigma, e \rangle)) = \langle \sigma', e' \rangle
\]
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

Notation: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$
which means $(\langle \sigma, e \rangle, \langle \sigma', e' \rangle) \in \rightarrow$.

Question: How should we define this relation?

$\langle \emptyset, 21 \times 2 \rangle \rightarrow \langle \sigma, 42 \rangle$
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

**Notation:** $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

which means $(\langle \sigma, e \rangle, \langle \sigma', e' \rangle) \in \text{“}\rightarrow\text{”}$.

**Question:** How should we define this relation? Remember that there are an infinite number of configurations and possible steps!
Inference Rules

**Answer:** Define it inductively, using *inference rules*:

\[
\begin{array}{c}
\text{premise}_1 \quad \text{premise}_2 \quad \cdots \\
\hline
\text{conclusion} \quad \text{NAME}
\end{array}
\]
Inference Rules

Answer: Define it inductively, using inference rules:

\[
\text{premise}_1 \quad \text{premise}_2 \quad \cdots \quad \text{Name} \\
\underline{\quad \text{conclusion}}
\]

An inference rule defines an implication: if all the premises hold, then the conclusion also holds.

Formally, “→” is the smallest relation that is closed under all the inference rules.
Variables

\[ n = \sigma(x) \]

\[ \langle \sigma, x \rangle \rightarrow \langle \sigma, n \rangle \quad \text{VAR} \]

\[ (x, n) \in \sigma \]

\[ 4 = \langle \{ (y, 4) \} \rangle \quad \text{VAR} \]

\[ \langle \{ (y, 4) \} \rangle \quad \rightarrow \quad \langle \ldots, 4 \rangle \quad \text{VAR} \]
Addition

\[ p = m + n \]
\[ \langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle \]

\[
\begin{align*}
4 + 2 \\
2 + 4 \\
2 + 2
\end{align*}
\]
Addition

\[ p = m + n \]
\[ \langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle \quad \text{ADD} \]

\[ \langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle \]
\[ \langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e'_1 + e_2 \rangle \quad \text{LADD} \]
Addition

\[
p = m + n
\]

\[
\langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle
\]

**ADD**

\[
\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle
\]

\[
\langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e'_1 + e_2 \rangle
\]

**LADD**

\[
\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e'_2 \rangle
\]

\[
\langle \sigma, n + e_2 \rangle \rightarrow \langle \sigma', n + e'_2 \rangle
\]

**RADD**

\[
\begin{array}{c}
(1 + 1) + (1 + 1) \\
\end{array}
\]

\[
2 + C(1+1)
\]

\[
2 + 2
\]

\[
4
\]
\[
p = m \times n \\
\langle \sigma, m \times n \rangle \rightarrow \langle \sigma, p \rangle
\]
Multiplication

\[ p = m \times n \]

\[ \langle \sigma, m \times n \rangle \rightarrow \langle \sigma, p \rangle \]

**MUL**

\[ \langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1 \rangle \]

**LMUL**

\[ \langle \sigma, e_1 \times e_2 \rangle \rightarrow \langle \sigma', e_1' \times e_2 \rangle \]

\[ \langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle \]

**RMUL**

\[ \langle \sigma, n \times e_2 \rangle \rightarrow \langle \sigma', n \times e_2' \rangle \]
Assignment

$$\sigma' = \sigma[x \mapsto n]$$

$$\langle \sigma, x := n ; e_2 \rangle \rightarrow \langle \sigma', e_2 \rangle$$  \text{ASSGN}

Notation: $\sigma[x \mapsto n]$ is a new function that mostly behaves like $\sigma$, except that it maps $x$ to $n$.

$$\langle \{ (y, 10) \}, \ y := 5 ; \ y + 2 \rangle$$

$$\rightarrow \langle \{ (y, 5) \}, \ y + 2 \rangle$$
Assignment

\[
\sigma' = \sigma[x \mapsto n] \\
\langle \sigma, x := n ; e_2 \rangle \rightarrow \langle \sigma', e_2 \rangle
\]

**Notation:** \( \sigma[x \mapsto n] \) is a *new* function that mostly behaves like \( \sigma \), except that it maps \( x \) to \( n \).

\[
\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle \\
\langle \sigma, x := e_1 ; e_2 \rangle \rightarrow \langle \sigma', x := e'_1 ; e_2 \rangle
\]

**ASSGN**
Operational Semantics

\[
\frac{n = \sigma(x)}{\langle \sigma, x \rangle \rightarrow \langle \sigma, n \rangle} \quad \text{VAR}
\]

\[
\frac{\langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e_1' + e_2 \rangle}{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle} \quad \text{LADD}
\]

\[
\frac{\langle \sigma, n + e_2 \rangle \rightarrow \langle \sigma', n + e_2' \rangle}{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle} \quad \text{RADD}
\]

\[
\frac{p = m + n}{\langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle} \quad \text{ADD}
\]

\[
\frac{\langle \sigma, e_1 * e_2 \rangle \rightarrow \langle \sigma', e_1' * e_2 \rangle}{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle} \quad \text{LMUL}
\]

\[
\frac{\langle \sigma, n * e_2 \rangle \rightarrow \langle \sigma', n * e_2' \rangle}{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle} \quad \text{RMUL}
\]

\[
\frac{p = m \times n}{\langle \sigma, m * n \rangle \rightarrow \langle \sigma, p \rangle} \quad \text{MUL}
\]

\[
\frac{\langle \sigma, x := e_1 ; e_2 \rangle \rightarrow \langle \sigma', x := e_1' ; e_2 \rangle}{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle} \quad \text{ASSGN1}
\]

\[
\frac{\sigma' = \sigma[x \mapsto n]}{\langle \sigma, x := n ; e_2 \rangle \rightarrow \langle \sigma', e_2 \rangle} \quad \text{ASSGN}
\]